Coding Basics and Scripts

Contents

1 Review

So far we have:

- learned how to load in packages
- learned how to create some plots with built in datasets

This week we will learn:

- Some basics of coding
- Basics of naming objects
- General principals of calling functions.

2 Coding basics

Last class we really jumped into the deep end. Today we're going to head back to the baby pool for a bit and learn some of the basics of using `R`.

First, `R` can be used as a calculator:

```r
1 / 200 * 30
```

```
## [1] 0.15
```

```r
(59 + 73 + 2) / 3
```

```
## [1] 44.66667
```

```r
sin(pi / 2)
```

```
## [1] 1
```

The calculator will follow the order of operations (exponents, multiplication/division, addition/subtraction) where you need to be mindful of how you use parentheses:

```
1/2 + 3
```

```
## [1] 3.5
```

```
1/(2+3)
```

```
## [1] 0.2
```

```
3*3^3
```

```
## [1] 81
```

```
(3*3)^3
```

```
## [1] 729
```

To create a new objects you can use the assignment operator `<-`:

```
x <- 3 * 4
x
```

```
## [1] 12
```

or the `=` sign:

```
x = 3 * 4
x
```

```
## [1] 12
```

You will make lots of assignments and `<-` is a pain to type. Don't be lazy and use `=`; it will work, but it will cause confusion later.

In RStudio you can use the keyboard shortcut – Alt + - (Alt **and** the minus sign) on a Windows machine or Opt –/- (option **and** the –/- key) on a Mac to make the `<-` sign.

Quick sidebar on types of objects

Note that I called `x` an object. That is the main term I will use for a general piece of *data* in this class. An object can contain much more than data though.

Last class, all of the objects we used were a `data.frame` or a `tibble`. For now we can consider these things to be equivalent. However, there are many other types of objects in `R`.

Most functions will work if your data is a `data.frame` or a `tibble` (some will only work if it is one of these quantities). So, most of the time that's what we'll be dealing with.

Other types of objects is `R`:

- atomic vector (what `x` above is)
- matrix (we'll discuss these later in the class)
- arrays (a bunch of matrices)
- list (can be a combination, of data frame's, matrices, vectors, etc.)

And we're back.

You can combine multiple elements into a vector with `c()`:

```
primes <- c(1, 2, 3, 5, 7, 11, 13)
```

And basic arithmetic is applied to every element of the vector:

```
primes * 2
```

```
## [1]  2  4  6 10 14 22 26
```

```
primes - 1
```

```
## [1]  0  1  2  4  6 10 12
```

```
primes^2
```

```
## [1]   1   4   9  25  49 121 169
```

Some easy way to create data are `:`, `seq`, and `rep`:

```
y <- 1:9
y
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

```
y2 <- seq(from = 1, to = 9, by = 1)
y2
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

```
y3 <- seq(from = 0, to = 1, by = 0.1)
y3
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
y32 <- seq(from = 0, to = 1, length.out = 9)
y32
```

```
## [1] 0.000 0.125 0.250 0.375 0.500 0.625 0.750 0.875 1.000
```

```
y4 <- rep(x = 1:3, times = 3)
y4
```

```
## [1] 1 2 3 1 2 3 1 2 3
```

```
y5 <- rep(x = 1:3, each = 3)
y5
```

```
## [1] 1 1 1 2 2 2 3 3 3
```

2.1 Comments

R will ignore any text after `#`. This allows to you to write **comments**, text that is ignored by R but read by other humans.

Comments should always be used in code. If you have any doubt that you will know exactly what a snippet of code does in 6 months, put a comment in.

You can always figure out *what* was done, you cannot always figure out *why* you did it. So the *why* is slightly more important than the *what*. However, your comments should give detail on both which will save you a lot of time.

Comments can be helpful for briefly describing what the subsequent code does.

```
# define primes
primes <- c(1, 2, 3, 5, 7, 11, 13)
# multiply primes by 2
primes * 2
```

```
## [1]  2  4  6 10 14 22 26
```

3 What's in a name?

Object names must start with a letter, and can only contain letters, numbers, `_` and `.` (a period).

You want your object names to be descriptive, so you'll need to adopt a convention for multiple words.

The textbook recommends **snake_case** where you separate lowercase words with `_`.

```
i_use_snake_case
otherPeopleUseCamelCase
some.people.use.periods
And_aFew.People_RENOUNCEconvention
```

3.1 Inspecting objects

You can inspect an object by typing its name:

```
x
```

```
## [1] 12
```

Note that the names in `R` are case sensitive

```
X
#> Error: object 'X' not found
```

R provides many functions to examine features of objects, for example

- `class()` - what kind of object is it (high-level)?
- `length()` - (for vectors) how long is it?
- `dim()` - (for matrix, array, dataframe) What are the dimensions?
- `attributes()` - does it have any metadata?
- `is.datatype()` - is the object of "*datatype*"? (i.e., `is.vector()`, `is.matrix()`, `is.data.frame()`, etc.)

```
x = 3 * 4
class(x)
```

```
## [1] "numeric"
```

```
length(x)
```

```
## [1] 1
```

```
dim(x)
```

```
## NULL
```

```
attributes(x)
```

```
## NULL
```

```
is.vector(x)
```

```
## [1] TRUE
```

```
is.matrix(x)
```

```
## [1] FALSE
```

```
is.data.frame(x)
```

```
## [1] FALSE
```

Let's check out some of these functions with the `cars` data:

```
data("cars")
class(cars)
```

```
## [1] "data.frame"
```

```
length(cars)
```

```
## [1] 2
```

```
attributes(cars)
```

```
## $names
## [1] "speed" "dist"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

```
is.vector(cars)
```

```
## [1] FALSE
```

```
is.matrix(cars)
```

```
## [1] FALSE
```

```
is.data.frame(cars)
```

```
## [1] TRUE
```

4 Calling functions and argument matching

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

Let's check out the function `matrix()` which can be used to create a matrix:

```
X <- matrix(data = 1:9, nrow = 3, ncol = 3,
            dimnames = list(c("A","B","C"), c("D","E","F")))
X
```

```
##   D E F
## A 1 4 7
## B 2 5 8
## C 3 6 9
```

```
is.matrix(X)
```

```
## [1] TRUE
```

(note: matrices are mostly used when you want to do matrix multiplication. They are not usually used to store data since data frame's are most useful.)

Argument matching is a set of rules that determine how `R` interprets function calls.

Generally, there are five types of argument matching:

- Exact matching: execute functions with exact argument

- Partial matching : execute functions with abbreviated argument
- Positional matching : execute functions solely based on position of the argument
- Mixed matching : execute functions with mixed argument types
- Dot-Dot-Dot matching: use of ellipses

4.1 Exact Argument Matching

The way I created `X` above is an example of exact argument matching. The arguements I used are:

- data= key in data
- nrow= # of row
- ncol= # of column
- dinames= row/column name

I could also create the same `X` with:

```
X <- matrix(dimnames = list(c("A","B","C"), c("D","E","F")),
            ncol = 3, nrow = 3,
            data = 1:9)
X
```

```
##   D E F
## A 1 4 7
## B 2 5 8
## C 3 6 9
```

With exact argument matching the ordering of the arguments does not matter.

The nice thing about doing this in RStudio is that it tells you the arguments when you initially write the function. To test this out write `matrix()` in the console and you can see it's arguments.

4.2 Partial Argument Matching

In here, we use the following partial argument tags to call the function

- nr= # of rows
- di= dinames
- nc= # of columns
- dat= data

```
X <- matrix(dat = 1:9, nr = 3, nc = 3,
            di = list(c("A","B","C"), c("D","E","F")))
X
```

```
##   D E F
## A 1 4 7
## B 2 5 8
## C 3 6 9
```

Can we further shorten the name of `data`?

```
X <- matrix(d = 1:9, nr = 3, nc = 3,
            di = list(c("A","B","C"), c("D","E","F")))
# Error in matrix(d = 1:9, nr = 3, nc = 3, di = list(c("A", "B", "C"), c("D",  :
#  argument 1 matches multiple formal arguments
```

`R` can't tell `d` and `di` apart.

4.3 Positional Argument Matching

Here, we won't use any of the names. Just use the position that they are supposed to go into.

What positon are they supposed to go into? Well, we can check using RStudio (where it tells you the name and ordering of the arguments) or we can use the `args()` function:

```
args(matrix)
```

```
## function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
## NULL
```

Notice that there is a `byrow` arguments after `ncol`. We weren't using this argument before. Why? Because we were happy with the default option `FALSE`. Many arguments have default options within a given function. You don't have to specify these arguments unless you want to change their value.

Here we'll use positional arguments with `byrow=FALSE` (the default that we've been using):

```
X <- matrix(1:9, 3, 3, FALSE,
            list(c("A","B","C"), c("D","E","F")))
X
```

```
##   D E F
## A 1 4 7
## B 2 5 8
## C 3 6 9
```

Here we'll use positional arguments with `byrow=TRUE` which will switch the way the data are read into the matrix.

```
X_t <- matrix(1:9, 3, 3, TRUE,
            list(c("A","B","C"), c("D","E","F")))
X_t
```

```
##   D E F
## A 1 2 3
## B 4 5 6
## C 7 8 9
```

What if we left out the `byrow` argument?

```
X <- matrix(1:9, 3, 3,
            list(c("A","B","C"), c("D","E","F")))
# Error in matrix(1:9, 3, 3, list(c("A", "B", "C"), c("D", "E", "F"))) :
#   invalid 'byrow' argument
```

Positional matching is a shorter way to do things, but you have to get all of the orderings exactly correct.

4.4 Mixed Argument Matching

Pretty self explanatory, the combination of the above. For example,

```
X <- matrix(1:9, 3, 3,
            dimnames = list(c("A","B","C"), c("D","E","F")))
X
```

This is what I usually use.

4.5 Dot-Dot-Dot Matching

This is more of an advanced topic, but while we're on the how to call functions let's discuss it.

Dot-Dot-Dot Matching, is a type of arguement that you will see in functions. For example, the `data.frame` function is one we can use to create data frames. Let's look at it's arguements:

```
args(data.frame)
```

```
## function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
##     fix.empty.names = TRUE, stringsAsFactors = FALSE)
## NULL
```

The dot-dot-dot is used to match unspecified formal arguments of a function.

The main argument to `data.frame` is the data! The data are included in the function, but there are no formal arguments for it (i.e., there is no `data=` or anything).

As a result, the data are included in the function with their own names. These are names that we make up, so they can have any formal arguments (they change every time). Hence the need for the `...` convention.

Let's try it out.

```
df <- data.frame(X = 1:9, Y = c(1,1,1,2,2,2,3,3,3))
df
```

```
##   X Y
## 1 1 1
## 2 2 1
## 3 3 1
## 4 4 2
## 5 5 2
## 6 6 2
## 7 7 3
## 8 8 3
## 9 9 3
```

```
df2 <- data.frame(Seq = 1:9, Rep = c(1,1,1,2,2,2,3,3,3))
df2
```

```
##   Seq Rep
## 1   1   1
## 2   2   1
## 3   3   1
## 4   4   2
## 5   5   2
## 6   6   2
## 7   7   3
## 8   8   3
## 9   9   3
```

Note that to create a data frame the lengths have to be the same

```
df <- data.frame(X = 1:8, Y = c(1,1,1,2,2,2,3,3,3))
# Error in data.frame(X = 1:8, Y = c(1, 1, 1, 2, 2, 2, 3, 3, 3)) :
#   arguments imply differing number of rows: 8, 9
```

For other functions the `...` is used because there are **tons** of arguements that can go in there and they don't want to list them all.

For example, let's look at the arguments of `plot()` which can be used to create plots:

```
args(plot)
```

```
## function (x, y, ...)
## NULL
```

It looks so simple! Actually, there are maybe 100 different formal arguments that can be used.